UPPSALA
UNIVERSITET

# Best Practices I: Organising, Debugging and Profiling Python Code

2018-02-06

Advanced Scientific Programming with Python

# Starting Survey

socrative
UU1

How much Python experience do you have?

How much programming experience do you have?

How much time do you spend programming?

# The Zen Of Python (PEP 20)

```
In [1]: import this
```

The Zen of Python, by Tim Peters

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.

# The Zen Of Python

- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

# Code Organisation

- Many projects start with a random collection of files

```
icm-39-33:python-course-library filipe$ ls -l
total 2976
-rw-r-----  1 filipe  staff   151355 Apr 25 17:40 1FFK.ipynb
-rw-r-----  1 filipe  staff     3017 Apr 25 17:30 2space_brute_force.py
-rw-r--r--  1 filipe  staff       24 Apr 25 17:25 README.md
-rw-r-----  1 filipe  staff  1360292 Apr 25 17:43 center_of_mass_aliging.ipynb
icm-39-33:python-course-library filipe$
```

- Some Python scripts, some Jupyter notebooks, etc…
- The naming style could be better… (1FFK???)
- As soon as the project grows bigger things quickly get ugly

```python
import numpy as np
…

def FSC(F1, F2):
def radial_average(data):
def fun2(z, f1, f2):
def vector_align(cm11, cm12, cm21, cm22):
…
#center of mass f1

cm11 = np.array(ndimage.measurements.center_of_mass(f1))
print cm11

cm12 = ndimage.measurements.center_of_mass(f1[:,24:,:])
print cm12
cm12 = np.array((14.5, 24, 14.5))
print cm12

#center of mass f1

cm21 = np.array(ndimage.measurements.center_of_mass(f2))
print cm21

cm22 = ndimage.measurements.center_of_mass(f2[:12,15:,:])
cm22 = np.array((10.299, 16.954, 21.223))
#cm22 = np.array((11.299, 15.954, 22.223))
print cm22

angles = vector_align(cm11,cm12,cm21,cm22)
print angles[0]

f2 = ndimage.interpolation.rotate(f2, angles[3], axes=(0,1), reshape=False, mode='wrap')
f2 = ndimage.interpolation.rotate(f2, angles[1], axes=(1,2), reshape=False, mode='wrap')
```

```
fig = plt.figure()
ax = fig.add_subplot(121)
ax.imshow(np.abs(f1.sum(axis=2)), cmap='plasma')
ax1 = fig.add_subplot(122)
ax1.imshow(np.abs(f2.sum(axis=2)), cmap='plasma')
plt.show()

#fsc = FSC(f1, f2)
#print len(fsc)

fsc_2 = FSC(f1,f2)
# index/pixel in resolution
res = []
for i in range(1,26):
    res.append(1.24E-9/(2*((i*75E-6)/np.sqrt(((i*75E-6)**2+0.4**2)))))

print res

res_round = [ '%.1e' % elem for elem in res ]

fig = plt.figure(figsize=((15,5)))
ax = fig.add_subplot(121)
ax.plot(np.arange(len(res)),np.abs(fsc), color='b')
#ax.set_xticklabels(res_round)
ax.set_xlabel('pixels')
ax.axhline(1/np.exp(1), color='r', label='1/e')
ax.axvline(15, color='k', linestyle='-.', label='detector edge')
ax.axvline(21.21, color='k', linestyle='--', label='corner 2D detector')
ax.axvline(25, color='k', label='corner 3D image')
ax.legend(loc='upper right')
ax.set_title('before alignment')
…
```

# Code Organisation

- A lot of code outside of functions in the example
- Again ambiguous names (`fun2`)
- Many special values (`10.299, 16.954`, etc…)
- “Sparse” documentation

- Python includes **Classes**, **Modules** and **Packages** to help you organise you code

# Python Classes

- Classes are the key features of object-oriented programming.
- A class is a structure for representing an object and the operations that can be performed on the object.
- In Python a class can contain attributes (variables) and methods (functions).
- A class is defined almost like a function, but using the `class` keyword, and the class definition usually contains a number of class method definitions (a function in a class).
- Each class method should have an argument self as its first argument. This object is a self-reference.

# Python Classes

- Some class method names have special meaning, for example:
  - __init__: The name of the method that is invoked when the object is first created.
  - __str__ : A method that is invoked when a simple string representation of the class is needed, as for example when printed.
  - There are many more, see https://docs.python.org/3/reference/datamodel.html#special-method-names

```python
class Point:
    """
    Simple class for representing a point in a Cartesian coordinate system.
    """

    def __init__(self, x, y):
        """
        Create a new Point at x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))
```

# Python Classes

- To create a new instance of a class:

```
p1 = Point(0, 0) # this will invoke the __init__ method in the Point class
print(p1)         # this will invoke the __str__ method
```

- To invoke a class method in the class instance p:

```
p2 = Point(1, 1)
p1.translate(0.25, 1.5)
print(p1)
print(p2)
```

- Note that calling class methods can modify the state of that particular class instance, but does not effect other class instances or any global variables.

- That is one of the nice things about object-oriented design: code such as functions and related variables are grouped in separate and independent entities.

# Python Modules

- One of the most important concepts in good programming is to reuse code and avoid repetitions.
- Functions and classes with a well-defined purpose and scope
- Reuse instead of repeating similar code in different part of a program (modular programming)
- Greatly improved readability and maintainability
- Fewer bugs, easier to extend and debug/troubleshoot
- Python modules are a higher-level modular programming construct, where we can collect related variables, functions and classes in a module
- A python module is defined in a python file
- Accessible to other Python modules and programs using the `import` statement

```python
"""
Example of a python module. Contains a variable called my_variable,
a function called my_function, and a class called MyClass.
"""

my_variable = 0

def my_function():
    """
    Example function
    """
    return my_variable

class MyClass:
    """
    Example class.
    """

    def __init__(self):
        self.variable = my_variable

    def set_variable(self, new_value):
        """
        Set self.variable to a new value
        """
        self.variable = new_value

    def get_variable(self):
        return self.variable
```

mymodule.py

# Python Modules

- We can import the module mymodule into our Python program using import:
```
import mymodule
```
- Use `help(module)` to get a summary of what the module provides:
```
help(mymodule)
```

# Python Modules

```
Help on module mymodule:

NAME
    mymodule

FILE
    /Users/rob/Desktop/scientific-python-lectures/mymodule.py

DESCRIPTION
    Example of a python module. Contains a variable called my_variable,
    a function called my_function, and a class called MyClass.

CLASSES
    MyClass

    class MyClass
     |  Example class.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |
     |  get_variable(self)
     |
     |  set_variable(self, new_value)
     |      Set self.variable to a new value

FUNCTIONS
    my_function()
        Example function

DATA
    my_variable = 0
```

# Python Modules

```
In [1]: import mymodule

In [2]: mymodule.my_variable
Out[2]: 0

In [3]: mymodule.my_function()
Out[3]: 0

In [4]: my_class = mymodule.MyClass()
   ...: my_class.set_variable(10)
   ...: my_class.get_variable()
   ...:
Out[4]: 10
```

# Python Packages

- When you've got a large number of Python Modules, you'll want to organise them into packages.
- Packages are namespaces which contain multiple packages and modules themselves.
- They are simply directories, but with a twist.
- Each package in Python is a directory which **must** contain a special file called `__init__.py`.
- This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

# Steps To Create A Python Package

- Working with Python packages is really simple. All you need to do is:
  1. Create a directory and give it your package's name.
  2. Put your classes in it.
  3. Create a `__init__.py` file in the directory
- The `__init__.py` file is executed when the package is imported.
- It is typically used to import classes and modules from the package.

# Python Package Example

- In this example, we will create an **`animals`** package
- It contains two module files named **`mammals.py`** and **`birds.py`** with a class each.
- Step 1: Create the Package Directory
- Step 2: Add Modules

```
class Mammals:
    def __init__(self):
        ''' Constructor for this class. '''
        # Create some member animals
        self.members = ['Tiger', 'Elephant', 'Wild Cat']


    def printMembers(self):
        print('Printing members of the Mammals class')
        for member in self.members:
            print('\t%s ' % member)
```

# Python Package Example

```
class Birds:
    def __init__(self):
        ''' Constructor for this class. '''
        # Create some member animals
        self.members = ['Sparrow', 'Robin', 'Duck']


    def printMembers(self):
        print('Printing members of the Birds class')
        for member in self.members:
           print('\t%s ' % member)
```

- Step 3: Add the **`__init__.py`** file

```
from mammals import Mammals
from birds import Birds
```

- Step 4: Test your package! Create a **`test_animals.py`** file outside of the package

```
# Import classes from your brand new package
from animals import Mammals
from animals import Birds

# Create an object of Mammals class & call a method of it
myMammal = Mammals()
myMammal.printMembers()

# Create an object of Birds class & call a method of it
myBird = Birds()
myBird.printMembers()
```

# Coding Style

- It’s important to follow a consistent coding style
- Python has its own Style Guide in PEP 8

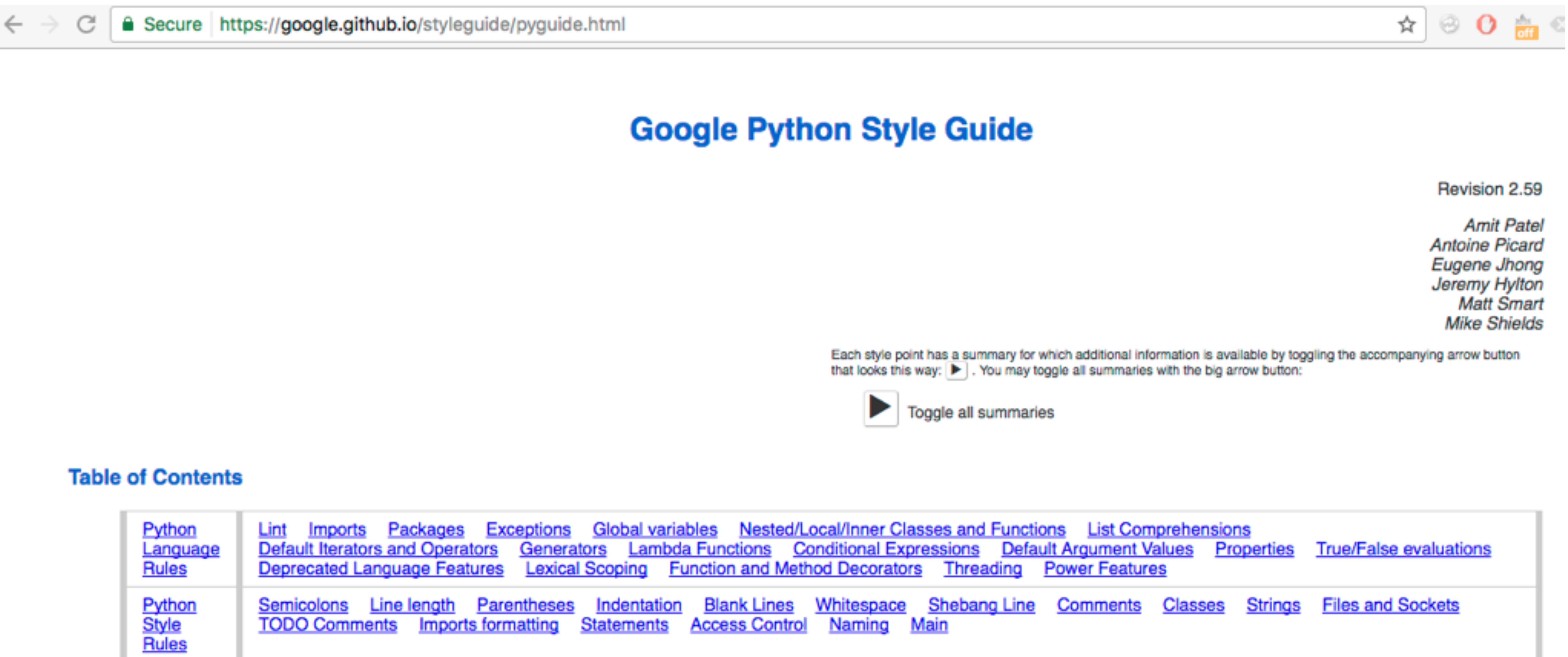
Python »» Python Developer's Guide »» PEP Index »» PEP 8 -- Style Guide for Python Code

## PEP 8 -- Style Guide for Python Code

| | |
|---|---|
| **PEP:** | 8 |
| **Title:** | Style Guide for Python Code |
| **Author:** | Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com> |
| **Status:** | Active |
| **Type:** | Process |
| **Created:** | 05-Jul-2001 |
| **Post-History:** | 05-Jul-2001, 01-Aug-2013 |

# Coding Style

- Another important one is the Google Python Style Guide


Secure https://google.github.io/styleguide/pyguide.html

## Google Python Style Guide

Revision 2.59

*Amit Patel*
*Antoine Picard*
*Eugene Jhong*
*Jeremy Hylton*
*Matt Smart*
*Mike Shields*

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▶ . You may toggle all summaries with the big arrow button:

▶ Toggle all summaries

### Table of Contents



- I recommend picking a coding style and sticking to it!

# Debugging

> It is a painful thing
> To look at your own trouble and know
> That you yourself and no one else has made it
>
> Sophocles, Ajax

- With a debugger, you can:
  - Explore the state of a running program
  - Test implementation code before applying it
  - Follow the program's execution logic
- You can set a **breakpoint** at any point of your program
- Much more powerful than using print() statements everywhere
- Being good at debugging is crucial to become a good programmer
- Python includes the **`pdb`**, the Python Debugger
- I prefer to use **`ipdb`**, which is similar but a bit more user friendly

## The buggy program

```
$ python main.py
Add the values
It's really that easy
Round
______________
2, 3, 4, 2, 5
______________
Sigh. What is your guess?: 16
Sorry that's wrong
The answer is: 5
Like seriously, how could you mess that up
Wins: 0 Loses 1
Would you like to play again?[Y/n]: y
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    main()
  File "main.py", line 8, in main
    GameRunner.run()
  File "/Users/filipe/Documents/Teaching/Advanced Scientific Programming
with Python/python-course/day1-basics/code/pdb-tutorial/dicegame/
runner.py", line 55, in run
    prompt = input("Would you like to play again?[Y/n]: ")
  File "<string>", line 1, in <module>
NameError: name 'y' is not defined
```

# ipdb In Action

- First, we have to import **ipdb**
- To analyse the code you need to set a breakpoint, **ipdb.set_trace()**
- Lets look at main.py:

```python
from dicegame.runner import GameRunner


def main():
    print("Add the values of the dice")
    print("It's really that easy")
    print("What are you doing with your life.")
    GameRunner.run()


if __name__ == "__main__":
    main()
```

Where you would you put the breakpoint?

socrative
UU1

# ipdb In Action

```python
from dicegame.runner import GameRunner


def main():
    print("Add the values of the dice")
    print("It's really that easy")
    print("What are you doing with your life.")
    import ipdb; ipdb.set_trace() # add pdb here
    GameRunner.run()


if __name__ == "__main__":
    main()
```

- Let's run main.py again and see what happens.

```
$ python main.py
Add the values of the dice
It's really that easy
What are you doing with your life.
> /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/python-course/day2-bestpractices-1/code/pdb-tutorial/main.py(9)main()
      8     import ipdb; ipdb.set_trace() # add pdb here
----> 9     GameRunner.run()
     10

ipdb>
```

# ipdb In Action

- We are now in the middle of the running program and we can start poking around.
- Top (i)pdb commands:

1. **`l(ist)`** - Displays 11 lines around the current line or continue the previous listing.
2. **`s(tep)`** - Execute the current line, stop at the first possible occasion.
3. **`n(ext)`** - Continue execution until the next line in the current function is reached or it returns.
4. **`b(reak)`** - Set a breakpoint (depending on the argument provided).
5. **`r(eturn)`** - Continue execution until the current function returns.
6. **`w(here)`** - Print a stack trace, with the most recent frame at the bottom.

# list - I'm too lazy to open the source code

```
l(ist) [first [,last]]

    List source code for the current file. Without arguments,
list 11 lines around the current line or continue the
previous listing. With one argument, list 11 lines starting
at that line. With two arguments, list the given range; if
the second argument is less than the first, it is a count.
```

- The above description was generated by calling `help` on `list`.
- Arguments specify range of lines you wish to see
- In Python 3.2 and above, `ll` (long list) shows the current function or frame

# list examples

```
(Pdb) l
  4     def main():
  5         print("Add the values of the dice")
  6         print("It's really that easy")
  7         print("What are you doing with your life.")
  8         import pdb; pdb.set_trace()
  9  ->     GameRunner.run()
 10
 11
 12     if __name__ == "__main__":
 13         main()
[EOF]

(Pdb) l 1, 13
  1     from dicegame.runner import GameRunner
  2
  3
  4     def main():
  5         print("Add the values of the dice")
  6         print("It's really that easy")
  7         print("What are you doing with your life.")
  8         import pdb; pdb.set_trace()
  9  ->     GameRunner.run()
 10
 11
 12     if __name__ == "__main__":
 13         main()
```

# `step` - let's see what this method does...

```
s(tep)

    Execute the current line, stop at the first possible
occasion (either in a function that is called or in the
current function).
```

```
(Pdb) l
  4     def main():
  5         print("Add the values of the dice")
  6         print("It's really that easy")
  7         print("What are you doing with your life.")
  8         import pdb; pdb.set_trace()
  9  ->     GameRunner.run()
 10
 11
 12     if __name__ == "__main__":
 13         main()

(Pdb) s
--Call--
> /Users/Development/pdb-tutorial/dicegame/runner.py(22)run()
-> @classmethod
```

# step examples

```
(Pdb) l
 17             total = 0
 18             for die in self.dice:
 19                 total += 1
 20             return total
 21
 22  ->     @classmethod
 23         def run(cls):
 24             # Probably counts wins or something.
 25             # Great variable name, 10/10.
 26             c = 0
 27             while True:

(Pdb) s
> /Users/Development/pdb-tutorial/dicegame/runner.py(26)run()
-> c = 0
(Pdb) l
 21
 22         @classmethod
 23         def run(cls):
 24             # Probably counts wins or something.
 25             # Great variable name, 10/10.
 26  ->         c = 0
 27             while True:
 28                 runner = cls()
 29
 30                 print("Round {}\n".format(runner.round))
 31
```

# next - I hope the current line doesn't throw an exception

```
n(ext)

    Continue execution until the next line in the current
function is reached or it returns.
```

# next examples

```
(Pdb) n
> /Users/Development/pdb-tutorial/dicegame/runner.py(27)run()
-> while True:
(Pdb) l
 22         @classmethod
 23         def run(cls):
 24             # Probably counts wins or something.
 25             # Great variable name, 10/10.
 26             c = 0
 27  ->         while True:
 28                 runner = cls()
 29
 30                 print("Round {}\n".format(runner.round))
 31
 32                 for die in runner.dice:
```

# next examples

```
(Pdb) n
> /Users/Development/pdb-tutorial/dicegame/runner.py(28)run()
-> runner = cls()
(Pdb) n
> /Users/Development/pdb-tutorial/dicegame/runner.py(30)run()
-> print("Round {}\n".format(runner.round))
(Pdb) n
Round 1

> /Users/Development/pdb-tutorial/dicegame/runner.py(32)run()
-> for die in runner.dice:
(Pdb) l
 27             while True:
 28                 runner = cls()
 29
 30                 print("Round {}\n".format(runner.round))
 31
 32  ->             for die in runner.dice:
 33                     print(die.show())
 34
 35                 guess = input("Sigh. What is your guess?: ")
 36                 guess = int(guess)
```

# break - I don't want to type n anymore

```
b(reak) [ ([filename:]lineno | function) [, condition] ]
    Without argument, list all breaks.

    With a line number argument, set a break at this line in the
    current file.  With a function name, set a break at the first
    executable line of that function.  If a second argument is
    present, it is a string specifying an expression which must
    evaluate to true before the breakpoint is honored.

    The line number may be prefixed with a filename and a colon,
    to specify a breakpoint in another file (probably one that
    hasn't been loaded yet).  The file is searched for on
    sys.path; the .py suffix may be omitted.
```

# break examples

- Setting a breakpoint

```
(Pdb) b 35
Breakpoint 1 at /Users/Development/pdb-tutorial/dicegame/runner.py(32)run()
(Pdb) c

[...] # prints some dice

> /Users/Development/pdb-tutorial/dicegame/runner.py(35)run()
-> guess = input("Sigh. What is your guess?: “)
```

- Viewing breakpoints

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/Development/pdb-tutorial/dicegame/runner.py:35
    breakpoint already hit 1 time
```

- Clear breakpoints

```
(Pdb) cl 1
Deleted breakpoint 1 at /Users/Development/pdb-tutorial/dicegame/runner.py:35
```

# return - I want to get out of this function

```
r(eturn)

    Continue execution until the current function returns.
(Pdb) l
 10         def reset(self):
 11             self.round = 1
 12             self.wins = 0
 13             self.loses = 0
 14
 15  ->     def answer(self):
 16             total = 0
 17             for die in self.dice:
 18                 total += 1
 19             return total
 20
(Pdb) r
--Return--
> /Users/Development/pdb-tutorial/dicegame/runner.py(19)answer()->5
-> return total
(Pdb) l
 14
 15         def answer(self):
 16             total = 0
 17             for die in self.dice:
 18                 total += 1
 19  ->         return total
 20
 21         @classmethod
 22         def run(cls):
 23             # Probably counts wins or something.
 24             # Great variable name, 10/10.
(Pdb)
```

# where - How did I get here

```
w(here)

   Print a stack trace, with the most recent frame at the bottom.
   An arrow indicates the "current frame", which determines the
   context of most commands.  'bt' is an alias for this command.
```

```
ipdb> w
  /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/python-course/day2-bestpractices-1/code/pdb-tutorial/main.py(13)<module>()
     11
     12 if __name__ == "__main__":
---> 13     main()

  /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/python-course/day2-bestpractices-1/code/pdb-tutorial/main.py(9)main()
      8     import ipdb; ipdb.set_trace() # add pdb here
----> 9     GameRunner.run()
     10

> /Users/filipe/Documents/Teaching/Advanced Scientific Programming with Python/python-course/day2-bestpractices-1/code/pdb-tutorial/dicegame/runner.py(32)run()
     31             for die in runner.dice:
---> 32                 print(die.show())
     33
```

# Arbitrary Python Commands

```
ipdb> l
     27                 runner = cls()
     28
     29                 print("Round {}\n".format(runner.round))
     30
     31                 for die in runner.dice:
---> 32                     print(die.show())
     33
     34                 guess = input("Sigh. What is your guess?: ")
     35                 guess = int(guess)
     36
     37                 if guess == runner.answer():

ipdb> die
<dicegame.die.Die instance at 0x1088b1998>
ipdb> die.value
5
ipdb> die.value = 1
ipdb> die.value
1
ipdb> die.show()
'---------\n|       |\n|   *   |\n|       |\n---------'
ipdb> print(die.show())
---------
|       |
|   *   |
|       |
---------
ipdb>
```

# Where Will Execution Stop?

```python
#!/usr/bin/python

def fact(x): return (1 if x==0 else x * fact(x-1))

def is_curious(n):
    s = str(n)
    sum = 0;
    for c in s:
        sum += fact(int(c))
    if(sum == n):
        return True
    return False

for a in range(10,1000000):
    import ipdb; ipdb.set_trace() # add pdb here
    if(is_curious(a)):
        print a
```

Where will the program stop after you start it?

And if you now do next?

And if you had done step instead of next?

And if you now do continue?

socrative
UU1

# Debugging Tips

- Fix the Problem, not the Blame
- Don't Panic
- Don't Assume It - Prove It

- Is the problem being reported a direct result of the underlying bug, or merely a symptom?
- If you had explained this problem in detail to a coworker, what would you say?
- If the suspect code passes its tests are the tests complete? What happens if you run them with ***this*** data?
- Do the conditions that caused this bug exist anywhere else in the system?

The Pragmatic Programmer

# Profiling Code

- Python provides the **`cProfile`** profiler as part of the standard library.
- cProfile is very simple to use, just:
  **`python -m cProfile script.py`**
- **Running it on the curious number script:**

```
$ python -m cProfile curious.py
145
40585
         33888828 function calls (6888873 primitive calls) in 10.921 seconds

   Ordered by: standard name

   ncalls            tottime  percall  cumtime  percall filename:lineno(function)
        1            0.205    0.205   10.921   10.921 profile.py:3(<module>)
32888835/5888880     6.589    0.000    6.589    0.000 profile.py:3(fact)
   999990            4.110    0.000   10.699    0.000 profile.py:5(is_curious)
        1            0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        1            0.016    0.016    0.016    0.016 {range}
```

# Profiling Code

```python
#!/usr/bin/python

def fact(x): return (1 if x==0 else x * fact(x-1))

def is_curious(n):
    s = str(n)
    sum = 0;
    for c in s:
        sum += fact(int(c))
    if(sum == n):
        return True
    return False

for a in range(10,1000000):
    import ipdb; ipdb.set_trace() # add pdb here
    if(is_curious(a)):
        print a
```

```
   ncalls             tottime  percall  cumtime  percall filename:lineno(function)
        1              0.205    0.205   10.921   10.921 profile.py:3(<module>)
32888835/5888880       6.589    0.000    6.589    0.000 profile.py:3(fact)
   999990              4.110    0.000   10.699    0.000 profile.py:5(is_curious)
        1              0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        1              0.016    0.016    0.016    0.016 {range}
```

# Line By Line Profiling

- In the previous example we just saw how long each function takes
- Often we want more fine grained knowledge
- The excellent `line_profiler` package provides this
- Install it with `pip install --user line_profiler`
- `line_profiler` comes with the kernprof script to help you run it
- You need to decorate the functions you want to profile with the `@profile` decorator
- `memory_profiler` is another package which provides line by line memory usage

# Line By Line Profiling

```python
#!/usr/bin/python

@profile
def fact(x): return (1 if x==0 else x * fact(x-1))

@profile
def is_curious(n):
    s = str(n)
    sum = 0;
    for c in s:
        sum += fact(int(c))
    if(sum == n):
        return True
    return False

@profile
def find_curious():
    for a in range(10,1000000):
        if(is_curious(a)):
            print a

find_curious()
```

# Line By Line Profiling

```
$ kernprof -l -v profile.py
145
40585
Wrote profile results to profile.py.lprof
Timer unit: 1e-06 s

Total time: 23.8517 s
File: profile.py
Function: fact at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4  32888835     23851734      0.7    100.0  def fact(x): return (1 if x==0 else x * fact(x-1))

Total time: 66.264 s
File: profile.py
Function: is_curious at line 6

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     6                                           @profile
     7                                           def is_curious(n):
     8    999990       455368      0.5      0.7      s = str(n)
     9    999990       283893      0.3      0.4      sum = 0;
    10   6888870      2270834      0.3      3.4      for c in s:
    11   5888880     62645645     10.6     94.5          sum += fact(int(c))
    12    999990       312672      0.3      0.5      if(sum == n):
    13         2            0      0.0      0.0          return True
    14    999988       295636      0.3      0.4      return False

Total time: 72.7165 s
File: profile.py
Function: find_curious at line 16

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           @profile
    17                                           def find_curious():
    18    999991       306494      0.3      0.4      for a in range(10,1000000):
    19    999990     72409951     72.4     99.6          if(is_curious(a)):
    20         2           48     24.0      0.0              print a
```

Times given in microseconds unless noted otherwise.

# Memory Profiling

- `memory_profiler` is another package which provides line by line memory usage
- Uses the same `@profile` decorator
- Install it with `pip install --user memory_profiler`
- You need to decorate the functions you want to profile with the `@profile` decorator
- You can run it as `python -m memory_profiler <script.py>`

# Memory Profiling

```
$ python -m memory_profiler profile.py
145
40585
Filename: profile.py

Line #    Mem usage    Increment   Line Contents
================================================
     4   58.152 MiB    0.000 MiB   @profile
     5   58.152 MiB    0.000 MiB   def fact(x): return (1 if x==0 else x * fact(x-1))


Filename: profile.py

Line #    Mem usage    Increment   Line Contents
================================================
     7   58.152 MiB    0.000 MiB   @profile
     8                             def is_curious(n):
     9   58.152 MiB    0.000 MiB       s = str(n)
    10   58.152 MiB    0.000 MiB       sum = 0;
    11   58.152 MiB    0.000 MiB       for c in s:
    12   58.152 MiB    0.000 MiB           sum += fact(int(c))
    13   58.152 MiB    0.000 MiB       if(sum == n):
    14   58.152 MiB    0.000 MiB           return True
    15   58.152 MiB    0.000 MiB       return False


Filename: profile.py

Line #    Mem usage    Increment   Line Contents
================================================
    17   27.047 MiB    0.000 MiB   @profile
    18                             def find_curious():
    19   27.051 MiB    0.004 MiB       time.sleep(10)
    20   58.152 MiB   31.102 MiB       for a in range(10,1000000):
    21   58.152 MiB    0.000 MiB           if(is_curious(a)):
    22   58.152 MiB    0.000 MiB               print a
```

# Profiling In IPython

- You can load memory_profiler and line_profiler in IPython with:
  ```
  %load_ext memory_profiler
  %load_ext line_profiler
  ```
- Now you'll have access to the magic commands `%lprun` and `%mprun` which behave similarly to their command-line counterparts.
- Except you won't need to decorate your functions with the `@profile` decorator.
- Just go ahead and run the profiling directly within your IPython session like so:
  ```
  In [1]: from primes import primes
  In [2]: %mprun -f primes primes(1000)
  In [3]: %lprun -f primes primes(1000)
  ```
- This can save you a lot of time and effort since none of your source code needs to be modified in order to use these profiling commands.

# Final Thoughts On Profiling

- Timing your code will change its timing
- Profiling incurs some performance penalty
- The finer the profiling the greater the penalty
- 10s with `cProfile` vs 70s with `line_profiler`
- **Always profile before optimising**

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."
>
> Donald Knuth

# References

Code examples have been take from

- https://github.com/jrjohansson/scientific-python-lectures/blob/master/Lecture-1-Introduction-to-Python-Programming.ipynb
- http://pythoncentral.io/how-to-create-a-python-package/
- https://www.learnpython.org/en/Modules_and_Packages
- https://github.com/spiside/pdb-tutorial
- http://lanyrd.com/2013/pycon/scdywg/
- https://www.huyng.com/posts/python-performance-analysis